ESD-TR-73-112

MULTIPLE EVALUATORS IN AN EXTENSIBLE PROGRAMMING SYSTEM

Ben Wegbreit

March 1973

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-71-C-0173 by Harvard University, Center for Research in Computing Technology, Cambridge, Mass. 02138.)

ESD-TR-73-112

MULTIPLE EVALUATORS IN AN EXTENSIBLE PROGRAMMING SYSTEM

Ben Wegbreit

March 1973

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-71-C-0173 by Harvard University,
Center for Research in Computing Technology, Cambridge, Mass. 02138.)

FOREWORD

This report was prepared in support of Project 2801, Task 02 by Harvard University, Cambridge, Massachusetts under USAF contract F19628-71-C-0173, monitored by Capt T J Rosenberger, Electronic Systems Division, MCIT, L G Hanscom Field, Bedford, Massachusetts, and was submitted 11 December 1972.

This technical report has been reviewed and is approved.

[illegible]

MELVIN B. EMMONS, Colonel, USAF
Director, Information Systems Technology
Deputy for Command & Management Systems

ABSTRACT

An effective tool for programming includes a high-level language, preferably extensible. A language alone is, however, not sufficient. One wants a complete programming system with an interpreter, a fully compatible compiler, a source level optimizer, and facilities for proving properties of programs. The purpose of this paper is to discuss how these various evaluators of the language can be fitted together and made to complement each other. The result, an extensible programming system with multiple evaluators, provides a powerful programming tool.

TABLE OF CONTENTS

Page

Introduction.. 1

Mixing Interpreted and Compiled Code................ 2

Unit Compilation.................................... 6

Assumptions and Assertions.......................... 9

Optimization, Extension Sets and Transformation Sets.. 13

Conclusion.. 20

References.. 21

Appendix: A Brief Description of EL1 Syntax........ 24

INTRODUCTION

As advanced computer applications become more complex, the need for good programming tools becomes more acute. The most difficult programming projects require the best tools. It is our contention that an effective tool for programming should have the following characteristics:

(1) Be a complete programming system – a language, plus a comfortable environment for the programmer (including an editor, documentation aids, and the like).

(2) Be extensible, in its data, operations, control, and interfaces with the programmer.

(3) Include an interpreter for debugging and several compilers for various levels of compilation – all fully compatible and freely mixable during execution.

(4) Include a program verifier that validates stated input/output relations or finds counter-examples.

(5) Include facilities for program optimization and tuning – aids for program measurement and a subsystem for automatic high-level optimization by means of source program transformation.

We will assume, not defend, the validity of these contentions here. Defenses of these positions by us and others have appeared in the literature.[1,2,3,4,5] The purpose of this paper is to discuss how these characteristics are to be simultaneously realized and, in particular, how the evaluators, verifier, and optimizer are to fit together. Compiling an extensible language where compiled code is to be freely mixed with interpreted code presents several novel problems and therefore a few unique opportunities for optimization. Similarly, extensibility and multiple evaluators make program automation by means of source level transformation more complex, yet provide additional handles on the automation process.

This paper is divided into five sections. Section 2 deals with communication between compiled and interpreted code, i.e., the runtime information structures and interfaces. Section 3 discusses one critical optimization issue in extensible languages – the compilation of unit operations. Section 4 examines the relation between debugging problems, proving the correctness of programs, and use of program properties in compilation. Finally, section 5 discusses the use of transformation sets as an adjunct to extension sets for application-oriented optimization.

Before treating the substantive issues, a remark on the implementation of the proposed solutions may be in order. Our acquaintance with these problems has arisen from our experience in the design, implementation, and use of the ECL programming system. ECL is an extensible programming system utilizing multiple evaluators; it has been operational on an experimental basis, running on a DEC PDP10, since August 1971. Some of the techniques discussed in this paper are functional, others are being implemented, still others are being designed. As the status of various points is continually changing, explicit discussion of their implementation state in ECL will be omitted.

For concreteness, however, we will use the ECL system and ECL's base language, EL1, as the foundation for discussion. An appendix treats those aspects of EL1 syntax needed for reading the examples in this paper.

MIXING INTERPRETED AND COMPILED CODE

The immediate problem in a multiple evaluator system is mixing code. A program is a set of procedures which call each other; some are interpreted, others compiled by various compilers which optimize to various levels. Calls and non-local gotos are allowed where either side may be either compiled or interpreted. Additionally, it is useful to allow control flow by means of RETFROM – that is the forced return from a specified procedure call (designated by name), with a specified value as if that procedure call had returned normally with the given value (cf. [6]).

Within each procedure, normal techniques apply. Interpreted code carries the data type of each entity – for autonomous temporary results as well as parameters and locals. Since the set of data types is open-ended and augmentable during execution, data types are implemented as pointers to (or indices in) the data type table. Compiled code can usually dispense with data types so that temporary results need not, in general, carry type information. In either interpreted or compiled procedures, where data types are carried, the type is associated not with the object but rather with a descriptor consisting of a type code and a pointer to the object. This results in significant economies whenever objects are generated in the free storage region.

Significant issues arise in communication between procedures. The interfaces must:

(1) Allow identification of free variables in one procedure with those of a lower access environment and supply data type information where required.

(2) Handle a special, but important, subcase of #1 – non-local gotos out of one procedure into a lower access environment.

(3) Check that the arguments passed to compiled procedure are compatible with the formal parameter types.

(4) Check that the result passed back to a compiled procedure (from a normal return of a called function or via a RETFROM) is compatible with the expected data type.

These communication issues are somewhat complicated by the need to keep the overhead of procedure interface as low as possible for common cases of two compiled procedures linking in desirable (i.e., well-programmed) ways.

The basic technique is to include in the binding (i.e., parameter block) for any new variable its name and its mode (i.e., its data type) in addition to its value. Names are implemented as pointers to (or indices in) the symbol table. (With reasonable restrictions on the number of names and modes, both name and mode can be packed into a 32-bit word.) Within a compiled procedure, all variables are referenced as a pair ⟨block level number, variable number within that block⟩. Translation from name to such a reference pair is carried out for each bound appearance of a variable during compilation; at run time, access is made using a display (cf. [7]). However, a free appearance of a variable is represented and identified by symbolic name. Connection between the free variable and some bound variable in an enclosing access environment is made during execution, implemented using either shallow or deep bindings (cf. [8] for an explanation of the issues and a discussion of the trade-offs for LISP). Once identification is made, the mode associated with the bound variable is checked against the expected mode of the free variable, if the expected mode is known.

To illustrate the last point, we suppose that in some procedure, P, it is useful to use the free variable BETA with the knowledge that in all correctly functioning programs the relevant bound BETA will always be a character string. To permit partial type checking during compilation, a declaration may be made at the head of the first BEGIN-END block of P.

```
DECL  BETA: STRING  SHARED  BETA;
```

This creates a local variable BETA of mode STRING which shares storage (i.e., binding by reference in FORTRAN or PL/I[9]) with the free variable BETA. All subsequent appearances of BETA in P are bound, i.e., identified with the local variable named BETA. Since the data type of the local BETA is known, normal compilation can be done for all internal appearances of BETA. The real identity of BETA is fixed during execution by identification with the free BETA of the access environment at the point P is entered. When the identification of bound and free BETA is made, mode checking (e.g., half-word comparison of two type codes) ensures that mode assumptions have not been violated.

In the worst case, parameter bindings entail the same sort of type checking. The arguments passed to a procedure come with associated modes. When a procedure is entered, the actual argument modes can be checked against the expected parameter modes and, where appropriate, conversion performed. Then the names of the formal parameters are added to the argument block, forming a complete variable binding. Notice that this works in all four cases of caller/callee pairs: compiled/compiled, compiled/interpreted, interpreted/compiled and interpreted/interpreted. Since type checking is implemented by a simple (usually half-word) comparison, the overhead is small.

However, for the most common cases of compiled/compiled pairs, mode checking is handled by a less flexible but more efficient technique. The mode of the called procedure may be declared in the caller. For example:

```
DECL  G: PROC(INT, STRING; COMPLEX) ;
```

specifies that G is a procedure-valued variable which takes two arguments, an integer and a character string, and returns a complex number. For each call on G in the range of this declaration, mode checking and insertion of conversion code can be done during compilation, with the knowledge that G is constrained to take on only certain procedure values. To guarantee this constraint, all assignments to (or bindings of) G are type checked. Type checking is made relatively inexpensive by giving G the mode PROC(INT, STRING; COMPLEX) – i.e., there is an entry in the data type table for it – and comparing this with the mode of the procedure value being assigned. The single comparison simultaneously verifies the validity of the result mode and both argument modes.

Result types are treated similarly. For each procedure call, a uniform call block is constructed* which includes the name of the procedure being called and the expected mode of the result (e.g., for the above example,

*This can be included in the LINK information.[7]

the name field is G and the expected-result-mode field is COMPLEX). This is ignored when compile-time checking of result type is possible and normal return occurs. However, if interpreted code returns to compiled code, or if RETFROM causes a return to a procedure by a non-direct callee, then the expected-result-mode field is checked against the mode of the value returned.

Transfer of control to non-local labels falls out naturally if labels are treated as named entities having constant value. On entry to a BEGIN-END block (in either interpreted or compiled code), a binding is made for each label in that block. The label value is a triple ⟨ indicator of whether the block is interpreted or compiled, program address, stack position⟩. A non-local goto label L is executed by identifying the label value referenced by the free use of L, restoring the stack position from the third component of the triple, and either jumping to the program address in compiled code or to the statement executor of the interpreter.

UNIT COMPILATION

In most programs the bulk of the execution time is spent performing the unit operations of the problem domain. In some cases (e.g., scalar calculations on reals), the hardware realizes the unit operations directly. Suppose, however, that this is not the case. Optimizing such programs requires recognizing instances of the unit operations and special treatment – unit compilation – to optimize these units properly.

An extensible language makes recognition a tractable problem, since the most natural style of programming is to define distinct data types for the unit entities, and procedures for the unit operations in each problem area. (Operator extension and syntax extension allow the invocation of these procedures by prefix and infix expressions and special statement types.) Hence, the unit operations are reasonably well-modularized. Detecting which procedures in the program are the critical unit operations entails static analysis of the call and loop structure, coupled with counts of call frequency during execution of the program over benchmark data sets.

The critical unit operations generally have one or more of the following characteristics:

(1) They have relatively short execution time; their importance is due to the frequency of call, not the time spent on each call.

(2) Their size is relatively small.

(3) They are terminal nodes of the call structure, or nearly terminal nodes.

(4) They entail a repetition, performing the same action over the lower-level elements which collectively comprise the unit object of the problem level.

Unit compilation is a set of special heuristics for exploiting these characteristics.

Since execution time is relatively small, call/return overhead is a significant fraction. Where the unit operations are terminal, the overhead can be substantially reduced. The arguments are passed from compiled code to a terminal unit operation with no associated modes. (Caller and callee know what is being transmitted.) The arguments can usually be passed directly in the registers. No bindings are made for the formal parameters. (A terminal node of the call structure calls no other; hence, there can be no free uses of these variables.) The result can usually be returned in a register, again, with no associated mode information.

Since the unit operations are important far out of proportion to their size, they are subject to optimizing techniques too expensive for normal application. Optimal ordering of a computation sequence (e.g., to minimize memory references or the number of temporary locations) can, in general,* be assured only by a search over a large number of possible orderings. Further, the use of identities (e.g., $a*b+a*c \rightarrow a*(b+c)$) to minimize the computational cost causes significant increase in the space of possibilities to be considered. The use of arbitrary identities, of course, makes the problem of program equivalence

*The only significant exception is for arithmetic expressions with no common subexpressions.[10]

(and, hence, of cost minimization) undecidable. However, an effective procedure for obtaining equivalent computations can be had either by restricting the sort of transformations admitted[11] or by putting a bound on the degree of program expansion acceptable. Either approach results in an effective procedure delivering a very large set of equivalent computations. While computationally intractable if employed over the whole program, a semi-exhaustive search of this set for the one with minimal cost is entirely reasonable to carry out on a small unit operator. Similarly, to take full advantage of multiple hardware function units, it is sometimes necessary to unwind a loop and rewind it with a modified structure – e.g., to perform, on the ith iteration of the new loop, certain computation which was formerly performed on the (i-1)st, ith, and (i+1)st iteration. Again, a search is required to find the optimal rewinding.

In general, code generation which tries various combinations of code sequences and chooses among them (by analysis or simulation) can be used in a reasonable time scale if consideration is restricted to the few unit operations where the pay-off is significant. Consider, for example, a procedure which searches through an array of packed k-bit elements counting the number of times a certain (parameter-specified) k-bit configuration occurs. The table can either be searched in array order – all elements in the first word, then all elements in the next, etc. – or in position order – all elements in the first position of a word, all elements in the next position, etc. Which search strategy is optimal depends on k, the hardware for accessing k-bit bytes from memory, the speed of shifting vs. memory access, and the sort of mask and comparison instructions for k-bit bytes. In many situations, the easiest way of choosing the better strategy is to generate code for each and compute the relative execution times as a function of array length.

A separate issue arises from non-obvious unit operations. Suppose analysis shows that procedures F and G are each key operations (i.e., are executed very frequently). It may well be that the appropriate candidates for unit compilation are F, G, and some particular combination of them, e.g., "F;G" or "G(...F(...)...)". That is, if a substantial number of calls on G are preceded by calls of F (in sequence or in an argument position),the new function defined by that composition should be unit compiled. For example, in dealing with complex arithmetic, +, –, *, /, and CONJ are surely unit operations. However, it may be that for some program, "u/v + v*CONJ(v)" is critical. Subjecting this combination to unit compilation saves four of the ten multiplications as well as a number of memory references.

ASSUMPTIONS AND ASSERTIONS

If an optimizing compiler is to generate really good code, it must be supplied the same sort of additional information that would be given to or deduced by a careful human coder. Pragmatic remarks (e.g., suggestions that certain global optimizations are possible) as well as explicit consent (e.g., the REORDER attribute of PL/I) are required. Similarly, if programs are to be validated by a program verifier, assistance from the programmer in forming inductive assertions is needed. Communication between the programmer and the optimizer/verifier is by means of ASSUME and ASSERT forms.

An assumption is stated by the programmer and is (by and large) believed true by the evaluator. A local assumption

```
ASSUME(X ≥ 0);
```

is taken as true at the point it appears. A global assumption may be extended over some range by means of the infix operator IN, e.g.,

```
ASSUME(X ≥ 0)   IN   BEGIN  ...  END;
```

where the assumption is to hold over the BEGIN-END block and over all ranges called by that block. The function of an assumption is to convey information which the programmer knows is true but which cannot be deduced from the program. Specifications of the well-formedness of input data are assumptions as are statements about the behavior of external procedures analyzed separately.

Assertions, on the other hand, are verifiable. From the program text and the validity of the program's assumptions, it is possible – at least in principle – to validate each assertion. For example,

```
ASSERT(FOR I FROM 1 TO N DO TRUEP(A[I] ≥ B[I])) IN BEGIN ... END
```

should be provably true over the entire BEGIN-END block, given that all program assumptions are correct.

The interpreter, optimizer, and verifier each treat assumptions and assertions in different ways. Since the _interpreter_ is used primarily for debugging, it takes the position that the programmer is not to be trusted. Hence, it checks everything, treating assumptions and assertions identically – as extended Boolean expressions to be evaluated and checked for _true_ (_false_ causing an ERROR and, in general, suspension of the program). Local assertions and assumptions are evaluated in analogy with the conditional expression

$$\text{NOT} \langle \text{expression} \rangle \Rightarrow \text{ERROR}(\ldots)$$

(This is similar to the use of ASSERT in ALGOL W.[12]) Assumptions and assertions over some range are checked over the entire range. This can be done by checking the validity at the start of the domain and setting up a _condition monitor_ (e.g., cf. [13]) which will cause a software interrupt if the condition is ever violated during the range.

Hence, in interpreted execution, assumptions and assertions act as comments whose correctness is checked by the evaluator, providing a rather nice debugging tool. Not only are errors explicitly detected by a false assertion, but when errors of other sorts occur (e.g., overflow, data type mismatch, etc.),the programmer scanning through the program is guaranteed that certain assertions were valid for that execution. Since debugging is often a matter of searching the execution path for the least source of an error, certainty that portions of the program are correct is as valuable as knowledge of the contrary.

The _compiler_ simply believes assertions and assumptions and uses their validity in code optimization. Consider, for example, the assignment

```
X ← B[I-J] - 60
```

Normally, the code for this would include subscript bounds checking. However, in

```
X ← (ASSERT(1 ≤ I-J ∧ I-J ≤ LENGTH(B)) IN B[I-J]) - 60
```

the assertion guarantees that the subscript is in range and no run-time check is necessary.

While assertions and assumptions are handled by the compiler in rather the same way, there are a few differences. Assumptions are the more powerful in that they can be used to express knowledge of program behavior which could not be deduced by the compiler, either because necessary information is not available (e.g., facts about a procedure which will be input during program execution) or because the effort of deduction is prohibitive (e.g., the use of deep results of number theory in a program acting on integers). Separate compilation makes the statement of such assumptions essential, e.g.,

```
ASSUME(SAFE(P)) IN BEGIN ... END
```

insures that the procedure P is free of side effects and hence can be subject to common subexpression elimination.

Unlike assumptions, assertions can be generated by the compiler as logical consequences of assumptions, other assertions, and the program text. Consider, for example, the following conditional block (cf. Appendix for syntax), where L is a pointer to a list structure.

```
BEGIN  L=NIL ⇒ ... ; ... CDR(L) ... END
```

Normally, the CDR operation would require a check for the empty list as an argument. However, provided that there are no intervening assignments to L, the compiler may rewrite this as

```
BEGIN  L=NIL ⇒ ... ; ASSERT(L≠NIL) IN  BEGIN ... CDR(L) ... END  END
```

in which case no checks are necessary. Assertions added by the compiler and included in an augmented source listing provide a means for the compiler to record its deductions and explicitly transmit these to the programmer.

The program <u>verifier</u> treats assumptions and assertions entirely differently.

Assumptions are believed.* Assertions are to be proved or disproved[14,15] on the basis of the stated assumptions, the program text, the semantics of the programming language, and specialized knowledge about the subject data types. In the case of integers, there has been demonstrable success – the assertion verifier of King has been applied successfully to some definitely non-trivial algorithms. Specialized theorem provers for other domains may be constructed. Fortunately, the number of domains is small. In ALGOL 60, for example, knowledge of the reals, the integers, and Boolean expressions together with an understanding of arrays and array subscripting will handle most program assertions.

*One might, conceivably, check the internal consistency of a set of assumptions, i.e., test for possible contradictions.

In an extensible language, the situation is more complex, but not drastically so. The base language data types are typically those of ALGOL 60 plus a few others, e.g., characters; the set of formation rules for data aggregates consists of arrays, plus structures and pointers. Only the treatment of pointers presents any new issues – these because pointers allow data sharing and hence access to a single entity under a multiplicity of names (i.e., access paths). This is analogous to the problem of subscript identification, but is compounded since the access paths may be of arbitrary length. However, recent work[16] shows promise of providing proof techniques for pointers and structures built of linked nodes. Since all extension sets ultimately derive their semantics from the base language, it suffices to give a formal treatment to the primitive modes and the builtin set of formation rules – assertions on all other modes can be mapped into and verified on the base.**

**This gives only a formal technique for verification, i.e., specifies what must be axiomatized and gives a valid reduction technique. It may well turn out that such reduction is not a practical solution if the resulting computation costs are excessive. In such cases, one can use the underlying axiomatization as a basis for deriving rules of inference on an extension set. These may be introduced in a fashion similar to the specialized transformation sets discussed in the next section.

One variation on the program verifier is the notifier. Whereas the verifier uses formal proof techniques to certify correctness, the notifier uses relatively unsophisticated means to provide counterexamples. One can safely assume that most programs will not be initially correct; hence, substantial debugging assistance can be provided by simply pointing out errors. This can be done somewhat by trial and error – generating values which satisfy the assumptions and running the program to check the assertions. Since programming errors typically occur at the extremes in the space of data values, a few simple heuristics may serve to produce critical counterexamples. If, as appears likely, the computation time for program verification is considerable, the use of a simple, quicker means to find the majority of bugs will be of assistance on online program production. While the notifier can never validate programs, it may be helpful in creating them.

OPTIMIZATION, EXTENSION SETS, AND TRANSFORMATION SETS

One of the advantages of an extensible language over a special purpose language developed to handle a new application arises from the economics of optimization. In an extensible language system, each extended language L_i is defined by an extension set E_i in terms of the base language. Since there is only a single base, one can afford to spend considerable effort in developing optimization techniques for it. Algorithms for register allocation, common subexpression detection, elimination of variables, removal of computation from loops, loop fusion, and the like need be developed and programmed only once. All extensions will take advantage of these. In contrast, the compiler for each special purpose language must have these optimizations explicitly included. This is already a reasonably large programming project, so large that many special purpose languages go essentially unoptimized. As the set of known optimization techniques grows, the economic advantage of extensible language optimization will increase.

There is one flaw in the above argument, which we now repair. There is the tacit assumption that all optimization properties of an extended language L_i can be obtained from the semantics and pragmatics of the base. While the logical dependency is strictly valid, taking this as a complete technique is rather impractical. While certain optimization properties — those concerned solely with control and data flow — can be well optimized in terms of the base language, othe properties depending on long chains of reasoning would tax any optimizer that sought to derive them every time they were required.

The point, and our solution, may best be exhibited with an example. Conside

FOO(SUBSTRING(I, J, X CONCAT Y))

which calls procedure FOO with the substring consisting of the I^th to (I+J-1)^th character of the string obtained by concatenating the contents of string variable X with string variable Y. In an extensible language, SUBSTRING and CONCAT are defined procedures which operate on STRINGs (defined to be ARRAYs of CHARacters).

```
SUBSTRING ←
EXPR(I,J:INT, S:STRING; STRING)
BEGIN
   DECL SS:STRING SIZE J;
   FOR K TO J DO SS[K] ← S[I+K-1];
   SS
END


CONCAT ←
EXPR(A,B:STRING; STRING)
BEGIN
   DECL R:STRING SIZE LENGTH(A)+LENGTH(B);
   FOR M TO LENGTH(A) DO R[M] ← A[M];
   FOR M TO LENGTH(B) DO R[M+LENGTH(A)] ← B[M];
   R
END
```

One could compile code for the above call on FOO by compiling three successive calls – on CONCAT, SUBSTRING, and FOO. However, by taking advantage of the properties of CONCAT and SUBSTRING, one can do far better. Substituting the definition of CONCAT in SUBSTRING produces

```
SUBSTRING(I, J, A CONCAT B) =
BEGIN
  DECL SS:STRING SIZE J;
  DECL S:STRING BYVAL
    BEGIN
      DECL R:STRING SIZE LENGTH(A)+LENGTH(B);
      FOR M TO LENGTH(A) DO R[M] ← A[M];
      FOR M TO LENGTH(B) DO R[M+LENGTH(A)] ← B[M];
      R
    END;
  FOR K TO J DO SS[K] ← S[I+K-1];
  SS
END
```

The block which computes R may be opened up so that its declarations and computation occur in the surrounding block. Then, since S is identical to R, S may be systematically replaced by R and the declaration for S deleted.

```
BEGIN
  DECL SS:STRING SIZE J;
  DECL R:STRING SIZE LENGTH(A)+LENGTH(B);
  FOR M TO LENGTH(A) DO R[M] ← A[M];
  FOR M TO LENGTH(B) DO R[M+LENGTH(A)] ← B[M];
  FOR K TO J DO SS[K] ← R[I+K-1];
  SS
END
```

This implies that R[M] is defined by the conditional block

```
BEGIN
  M ≤ LENGTH(A) ⇒ A[M];
  B[M-LENGTH(A)]
END
```

Replacing M by I+K-1 and substituting, the assignment loop becomes

```
FOR K TO J DO SS[K] ← BEGIN
                          K ≤ LENGTH(A)-I+1 ⇒ A[I+K-1];
                          B[I+K-LENGTH(A)-1]
                      END
```

Distributing the assignment to inside the block, this has the form

```
FOR x TO $v_0$ DO BEGIN
                    x ≤ $v_1$ ⇒ $f_1$(x);
                    $f_2$(x)
                  END
```

where v_i are loop-independent values and f_i are functions in x. A basic optimization on the base language transforms this into the equivalent form which avoids the test

```
FOR x TO MIN($v_0$,$v_1$) DO $f_1$(x);
FOR x FROM MIN($v_0$,$v_1$)+1 TO $v_0$ DO $f_2$(x);
```

Hence,SUBSTRING(I, J, A CONCAT B) may be computed by a call on the procedure*

```
EXPR(I,J:INT, A,B:STRING; STRING)
BEGIN
  DECL SS:STRING SIZE J;
  FOR K TO MIN(J, LENGTH(A)-I+1) DO S[K] ← A[I+K-1];
  FOR K FROM MIN(J, LENGTH(A)-I+1)+1 TO J DO S[K] ← B[I+K-LENGTH(A)-1];
  SS
END
```

*Normal common subexpression elimination will recognize that LENGTH(A), I-1, and MIN(J,LENGTH(A)-I+1) need be calculated only once.

This could, in principle, be deduced by a compiler from the definitions of SUBSTRING and CONCAT. However, there is no way for the compiler to know a priori that the substitution has substantial payoff. If the expression SUBSTRING(I,J,A CONCAT B) were a critical unit operation, the heuristic "try all possible compilation techniques on key expressions" would discover it. However, the compiler cannot afford to try all function pairs appearing in the program in the hope that some will simplify — the computational cost is too great. Instead, the programmer specifies to the compiler the set of transformations (cf. [17] for related techniques) he knows will have payoff.

```
    TRANSFORM(I,J:INT,  X,Y:STRING;  SUBSTITUTE)
          SUBSTRING(I, J, X CONCAT Y)
    TO
          SUBSTITUTE(Z:X CONCAT Y,  SUBSTRING(I,J,Z))  (I, J, X, Y)
```

In general, a transformation rule has the format

```
        TRANSFORM(⟨pattern variables⟩; ⟨action variables⟩)
              ⟨pattern⟩
        TO
              ⟨replacement⟩
```

All lexemes in the pattern and replacement are taken literally except for the ⟨pattern variables⟩ and ⟨action variables⟩. The former are dummy arguments, statement-matching variables, etc.; the latter denote values used to derive the actual transformation from the input transformation schemata. In the above case, the procedure SUBSTITUTE is called to expand CONCAT within SUBSTRING as the third argument. The simplified result, $\mathscr{P}$, is applied to the dummy arguments. Hence, calls such as SUBSTRING(3,2*N+C, AA CONCAT B7) are transformed into calls on $\mathscr{P}$(3,2*N+C, AA, B7)

When defining an extension set, the programmer defines the unit data types, unit operations, and additionally the significant transformations on the problem domain. These domain-dependent transformations are adjoined to the set of base transformations to produce the total transformation set. The program, as written, specifies the function to be computed; the transformation set provides an orthogonal statement of how the computation is to be optimized.

For example, in adding a string manipulation extension, one would first define the data type STRING (fixed length array of characters). Next,one defines the unit operations: LENGTH, CONCATenate, SUBSTRING, SEARCH (for a string x as part of a string y starting at position i and return the initial index or zero if not present). Finally, one defines the transformations on program units involving these operations.

```
TRANSFORM(X,Y:STRING)   LENGTH(X CONCAT Y)
TO   LENGTH(X)+LENGTH(Y)

TRANSFORM(A,X,Y,Z:STRING; SUBSTITUTE)  X  CONCAT  Y  CONCAT  Z
TO   SUBSTITUTE(A: Y CONCAT Z; X CONCAT A)  (X,Y,Z)
```

So long as the transformations are entirely local, they act only as macro replacements. The significant transformations in an extension set are those which make global, far reaching changes to program or data. Clearly, these changes will require knowledge, assumed or asserted, about that portion of the program affected by these changes.

Consider, for example, the issue of string variables in the proposed extensior set. If a string variable is to have a fixed capacity, the type STRING is satisfactoı If variable capacity is desired but an upper bound can be established for each strin variable, the type VARSTRING could be defined like string VARYING in PL/I. If completely variable capacity is required, a string variable would be implemented as a pointer to a simple STRING (i.e., PTR(STRING)) with the understanding that assignment of a string value to such a string variable causes a copy of the string to be made and the pointer set to address the copy.* With these three possible representations available, one would define the data type <u>string variable</u> to be

ONEOF(STRING, VARSTRING, PTR(STRING))

Each string variable is one of these three data types. To provide for the worst case, the programmer could specify each formal parameter string variable to be

---

*This does not exhaust the list of possible representations for strings. To avoid copying in concatenation, insertion, and deletion, one could represent strings by linked lists of characters nodes: each node consisting of a character and a pointer to the next node. A string variable could then be a pointer to such node lists. To minimize storage, one could employ hashing to insure that each distinct sequence of characters is represented by a unique string-table-entry; a string variable could then be a pointer to such string-table-entries. Hashing and implementing strings by linked lists could be combined to yield still another representation of strings. In the interest of brevity, we consider only three rather simple representations; however, the point we make is all the stronger when additional representations are considered.

ONEOF(STRING, VARSTRING, PTR(STRING)) and specify each local string variable to be a PTR(STRING). A program so written would be correct, but its performance would, in general, suffer from unused generality. Each string variable whose length is fixed can be redeclared

```
TRANSFORM(D1,D2:DECLIST, S:STATLIST, F:FORM, X; WHEN)
      WHEN (CONSTANT(LENGTH(X))) IN
         BEGIN D1; DECL X:PTR(STRING) BYVAL F; D2; S END
TO
      BEGIN D1; DECL X:STRING BYVAL F; D2; S END
```

The predicate WHEN appearing in a pattern is handled in somewhat the same fashion as are ASSERTions during program verification. It is proved as part of the pattern matching; the transformation is applicable only if the predicate is provably TRUE and the literal part of the pattern matches. Here, it must be proved that LENGTH(X) is a constant over the block B and all ranges called by B. If so, the variable can be of type STRING. Similarly, if there is a computable maximum length less than a reasonable upper limit LIM, then the data type VARSTRING can be used.

```
TRANSFORM(D1,D2:DECLIST, B:BLOCK, F:FORM, K:INT, X; WHEN)
      BEGIN D1; DECL X:PTR(STRING) BYVAL F; D2;
         WHEN(LENGTH(X) ≤ K ∧ K ≤ LIM) IN B
      END
TO
      BEGIN D1; DECL X:VARSTRING SIZE K BYVAL F; D2; B END
```

To prove an assertion for a variable X over some range, it suffices to prove the assertion true of all expressions that are assignable to X in that range. An assertion about LENGTH(X) is reasonable to validate since it entails only theorem proving over the integers[18] – once the string manipulation routines are reinterpreted as operations on string lengths. Fortunately, most of the interesting predicates are of this order of difficulty. Typical WHEN conditions are: (1) a variable (or certain fields of a data structure) is not changed; (2) an object in the heap is referenced only from a given pointer; (3) whenever control reaches a given program point, a variable always has (or never has) a given value (or set of values); (4) certain operations are never performed on certain elements of a data structure. Such conditions are usually easier to check than those concerned with correct program behavior, since only part of the action carried out by the algorithm is relevant.

That is, the technique suggested above for simplifying proofs about string manipulation operators by considering only string lengths generalizes to many related cases. To verify a predicate concerned with certain properties, one takes a valuation of the program on a model chosen to abstract those properties.[19] The program is run by a special interpreter which performs the computation on the simpler data space tailored to the property. To correct for the loss of information (e.g., the values of most program tests are not available), the computation is conservative (e.g., the valuation of a conditional takes the union of the valuations of the possible arms). If the valuation in the model demonstrates the proposition, it is valid for the actual data space. While this is a sufficient condition, not a necessary one, an appropriate model should seldom fail to prove a true proposition.

CONCLUSION

An interpreter, a compiler, a source-level optimizer employing domain-specific transformations, and a program verifier each compute a valuation over some model. Fitting these valuators together so as to exploit the complementarity of their models is a central task in constructing a powerful programming tool.

ACKNOWLEDGMENT

The author would like to thank Glenn Holloway and Richard Stallman for discussions concerning various aspects of this paper.

REFERENCES

1 B WEGBREIT

The ECL programming system

Proc AFIPS 1971 FJCC Vol 39 AFIPS Press Montvale New Jersey pp 253-262

2 A J PERLIS

The synthesis of algorithmic systems

JACM Vol 17 No 1 January 1967 pp 1-9

3 T E CHEATHAM *et al.*

On the basis for ELF – an extensible language facility

Proc AFIPS FJCC 1968 Vol 33 pp 937-948

4 D G BOBROW

Requirements for advanced programming systems for list processing

CACM Vol 15 No 7 July 1972

5 T E CHEATHAM and B WEGBREIT

A laboratory for the study of automating programming

Proc AFIPS 1972 SJCC Vol 40

6 W TEITELMAN *et al.*

BBN-LISP

Bolt Beranek and Newman Inc Cambridge Massachusetts July 1971

7 E W DIJKSTRA

Recursive programming

Numerische Mathematik 2 (1960) pp 312-318. Also in Programming Systems and Languages S Rosen (Ed) McGraw-Hill New York 1967

8 J MOSES

The function of FUNCTION in LISP

SIGSAM Bulletin July 1970 pp 13-27

9 IBM SYSTEM/360

PL/I language reference manual

Form C28-8201-2 IBM 1969

10 R SETHI and J D ULLMAN

The generation of optimal code for arithmetic expressions

JACM Vol 17 No 4 October 1970 pp 715-728

11 A V AHO and J D ULLMAN

Transformations on straight line programs

Conf Rec Second Annual ACM Symposium on Theory of Computing SIGACT May 1970 pp 136-148

12 R L SITES

Algol W reference manual

Technical Report CS-71-230 Computer Science Department Stanford University August 1971

13 D G BOBROW and B WEGBREIT

A model and stack implementation of multiple environments

Report No 2334 Bolt Beranek and Newman Cambridge Massachusetts March 1972 submitted for publication

14 R F FLOYD

Assigning meanings to programs

Proc Symp Appl Math Vol 19 1967 pp 19-32

15 R F FLOYD

Toward interactive design of correct programs

Proc IFIP Congress 1971 Ljubljana pp 1-5

16 J POUPON and B WEGBREIT

Verification techniques for data structures including pointers

Center for Research in Computing Technology Harvard University in preparation

17 B A GALLER and A J PERLIS

A proposal for definitions in Algol

CACM Vol 10 No 4 April 1967 pp 204-219

18 J C KING

A program verifier

PhD Thesis Department of Computer Science Carnegie-Mellon University September 1969

19 M SINTZOFF

Calculating properties of programs by valuations on specific models

SIGPLAN Notices Vol 7 No 1 and SIGACT News No 14 January 1972 pp 203-207

20 B WEGBREIT *et al*.

ECL programmer's manual

Center for Research in Computing Technology Harvard University Cambridge Massachusetts January 1972

APPENDIX: A BRIEF DESCRIPTION OF EL1 SYNTAX

To a first approximation, the syntax of EL1 is like that of ALGOL 60 or PL/I. Variables, subscripted variables, labels, arithmetic and Boolean expressions, assignments, gotos and procedure calls can all be written as in ALGOL 60 or PL/I. Further, EL1 is – like ALGOL 60 or PL/I – a block structured language. Executable statements in EL1 can be grouped together and delimited by BEGIN END brackets to form blocks. New variables can be created within a block by declaration; the scope of such variable names is the block in which they are declared.

The syntax of EL1 differs from that of ALGOL 60 or PL/I most notably in the form of conditionals, declarations, and data type specifiers. For the purposes of this paper, it will suffice to explain only these points of difference. (A more complete description can be found in [20].)

A.1 Conditionals

Conditionals in EL1 are a special case of BEGIN END blocks. In general, each EL1 block has a value – the value of the last statement executed. Normally, this is the last statement in the block. Instead, a block can be conditionally exited with some other value $\mathscr{V}$ by a statement of the form

$$\mathscr{B} \Rightarrow \mathscr{V};$$

If $\mathscr{B}$ is TRUE then the block is exited with the value of $\mathscr{V}$; otherwise, the next statement of the block is executed. For example, the ALGOL 60 conditional

$$\underline{\text{if}}\ \mathscr{B}_1\ \underline{\text{then}}\ \mathscr{V}_1\ \underline{\text{else}}\ \underline{\text{if}}\ \mathscr{B}_2\ \underline{\text{then}}\ \mathscr{V}_2\ \underline{\text{else}}\ \mathscr{V}_3$$

is written in EL1 as

$$\text{BEGIN}\ \mathscr{B}_1 \Rightarrow \mathscr{V}_1;\ \mathscr{B}_2 \Rightarrow \mathscr{V}_2;\ \mathscr{V}_3\ \text{END}$$

(Unconditional statements of an EL1 block are simply executed sequentially – unless a goto transfers control to a different labeled statement.)

A.2 Declarations

The initial statements of a block may be declarations having the format

$$\text{DECL}\ \mathcal{L}\!:\ \mathscr{M}\ \mathcal{S};$$

where $\mathcal{L}$ is a list of identifiers, $\mathscr{M}$ is the data type, and $\mathcal{S}$ specifies the initialization. For example,

DECL X,Y: REAL BYVAL A[I];

This creates two REAL variables named X and Y and initializes them to separate <u>copies</u> of the current value of A[I]. The specification $\mathcal{S}$ may assume one of three forms:

(1) empty — in which case a default initialization determined by the data type is used.

(2) BYVAL $\mathcal{V}$ — in which case the variables are initialized to copies of the value of $\mathcal{V}$.

(3) SHARED $\mathcal{V}$ — in which case the variables are declared to be synonymous with the value of $\mathcal{V}$.

A.3 Data types

Builtin data types of the language include: BOOL, CHAR, INT, and REAL. These may be used as data type specifiers to create scalar variables.

Array variables may be declared by using the builtin procedure ARRAY. For example,

```
DECL C: ARRAY(CHAR) BYVAL 𝒱 ;
```

creates a variable named C which is an ARRAY of CHARacters. The LENGTH (i.e., number of components) and initial value of C is determined by the value of $\mathcal{V}$.

Procedure-valued variables may be defined by the builtin procedure PROC. For example,

```
DECL G:PROC(BOOL,ARRAY(INT); REAL);
```

declares G to be variable which can be assigned only those procedures which take a BOOL argument and an ARRAY(INT) argument and deliver a REAL result.

A.4 Procedures

A procedure may be defined by assigning a procedure value to a procedure-valued variable. For example,

```
IPOWER ←
EXPR(X:REAL,N:INT; REAL)
BEGIN DECL R:REAL BYVAL 1; FOR I TO N DO R ← R*X; R END
```

assigns to IPOWER a procedure which takes two arguments, a REAL and an INT (assumed positive), and computes the exponential.

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Harvard University Center for Research in Computing Technology Cambridge, Mass. 02138	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED 2b. GROUP N/A

3. REPORT TITLE
MULTIPLE EVALUATORS IN AN EXTENSIBLE PROGRAMMING SYSTEM

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
None

5. AUTHOR(S) *(First name, middle initial, last name)*
Ben Wegbreit

6. REPORT DATE March 1973	7a. TOTAL NO. OF PAGES 30	7b. NO. OF REFS 20

8a. CONTRACT OR GRANT NO. F19628-71-C-0173 b. PROJECT NO. 2801 Task 02 c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-73-112 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

An effective tool for programming includes a high-level language, preferably extensible. A language alone is, however, not sufficient. One wants a complete programming system with an interpreter, a fully compatible compiler, a source level optimizer, and facilities for proving properties of programs. The purpose of this paper is to discuss how these various evaluators of the language can be fitted together and made to complement each other. The result, an extensible programming system with multiple evaluators, provides a powerful programming tool.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Programming system						
Extensible						
Interpreter						
Compilers						
Program verifier						
Program Optimization and tuning						

Unclassified

Security Classification